AFRL-IF-RS-TR-2006-11
Final Technical Report
January 2006

REPAIRABLE ARCHITECTURE FOR INTRUSION-TOLERANT NETWORK-CENTRIC SYSTEMS

Stony Brook University

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

STINFO FINAL REPORT

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2006-11 has been reviewed and is approved for publication

APPROVED: /s/

JOSEPH J. GIORDANO
Project Engineer

FOR THE DIRECTOR: /s/

WARREN H. DEBANY JR., Technical Advisor
Information Grid Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	JANUARY 2006	Final May 2005 – Aug 2005

4. TITLE AND SUBTITLE
REPAIRABLE ARCHITECTURE FOR INTRUSION-TOLERANT NETWORK-CENTRIC SYSTEMS

5. FUNDING NUMBERS
C - FA8750-05-1-0232
PE - N/A
PR - N/A
TA - N/A
WU - N/A

6. AUTHOR(S)

Tzi-cker Chiueh

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Stony Brook University
Computer Science Department

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Research Laboratory/IFGB
525 Brooks Road
Rome New York 13441-4505

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

AFRL-IF-RS-TR-2006-11

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Joseph J. Giordano/IFGB/(315) 330-1518/ Joseph.Giordano@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 Words)***
In summary, given an internet service, Arachne automatically instruments its application components to keep track of inter-request dependencies and associations between incoming requests and persistent state update operations. In addition, Arachne keeps proper update logs to ensure every file system and DBMS update operation is undoable. At repair time, Arachne derives the set of requests that are directly or indirectly dependent on the error/attack request in question, initiates file systems and DBMS undo operations to erase the side effects of these requests, and finally re-executes them according to their timestamps to re-incorporate their effects into the state of the internet service.

14. SUBJECT TERMS
DBMS, Arachne, MTTR, MTTB

15. NUMBER OF PAGES
8

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Table of Contents

1 Introduction 1

2 Technical Approach 2

3 Impacts 4

Repairable Architecture for Intrusion-Tolerant Network-Centric Systems

Tzi-cker Chiueh

Computer Science Department
Stony Brook University
chiueh@cs.sunysb.edu

1 Introduction

Despite a great deal of research on computer/network security, there is no such thing as un-breakable systems. Social engineering is one class of attacks that exploit human weaknesses to gain unauthorized accesses to technically secure systems. Insider attack is another class that is almost impossible to defeat in real time with absolute certainty. The fact that it is impossible to anticipate and seal all security holes argues strongly for a *fast repair* approach to secure computer system design. That is, whereas most past computer security research concentrated on the design of better protection mechanisms to detect or prevent intrusion, it's time to accept that computer systems cannot be absolutely impenetrable, and thus shift at least some of the cybersecurity research focus to the development of system design techniques that can minimize the cost of computer security breach by facilitating post-intrusion system clean-up and repair.

In the context of computer security, a system's availability metric is defined as $\frac{MTTB}{MTTB+MTTR}$, where MTTB refers to mean time between security breaches and MTTR refers to mean time to repair after a breach. To maximize the availability metric, computer security system designers could either increase MTTB to infinity, essentially building an absolutely secure system, or decrease MTTR to zero, i.e., repairing an attacked system instantly. As in fault-tolerant systems, pushing the MTTB beyond a certain level may become exponentially more expensive for every fixed increment of improvement, e.g., from 0.9999 to 0.99999. Therefore the most cost-effective approach to designing intrusion-tolerant systems may well be a combination of MTTB maximization and MTTR minimization.

A security vulnerability is a system fault, which, when exploited, leads to a failure that manifests itself as an intrusion. Traditional fault-tolerant systems assume a fail-stop model, where the system is able to detect a failure and stop immediately after it occurs. While this model works reasonably well for hardware failures and to some extent software failures, it is inadequate for system failures due to innocent human errors or malicious attacks. In those cases, the *failure detection window*, which is the interval between when a failure occurs and when it is detected, could easily be hours if not days. Some of the side effects that take place during a failure detection window are considered corrupted because their values are data-dependent on the failure, whereas the rest are considered benign. For faults that result in long detection window, simply restoring the system back to the state before the failure takes place is unacceptable in many cases because it leads to too much collateral damage, i.e., many benign side effects are thrown away unnecessarily. Imagine an enterprise web-based service is compromised and the attack is detected only 24 hours later, how does the service's system administrator clean up the state corruption due to this intrusion? Today, the system administrator has two options. First, she can restore the service's persistent storage, i.e., the database and the

file system, back to the state before the input error took place. This approach is simple and fast, but suffers from the drawback that all non-corrupted updates during the 24-hour period are unnecessarily wiped out. Alternatively, she can attempt to preserve as much non-corrupted data as possible by manually removing all the side effects that are contaminated by the attack. However, this approach is labor-intensive, time-consuming and thus error-prone. In practice, neither option is desirable.

A *repairable* information system is one that after an error/attack can quickly and automatically erase corrupted side effects while preserving benign ones, all without human intervention. Standard database recovery mechanisms only aim to restore a failed system back to the state before the failure occurs, and thus are different from repair mechanisms, whose goal is to preserve as many benign side effects as possible after a failure is detected. The two fundamental issues in designing any repairable information system are:

1. How to maintain the before image of every state update so that each state update is undoable, and
2. How to keep track of dependencies among update operations to the shared state so that it is possible to identify those and only those updates that are affected by an error or attack?

Update logging is the standard solution to the first issue, and has been built into many information systems, e.g., transaction logs in DBMS servers and metadata journals in file systems. *Inter-operation dependency tracking*, on the other hand, is rarely supported in existing information systems, but actually plays a more important role, because in practice, the most time-consuming part of a post-intrusion repair process is to scope out the exact "damage perimeter" associated with an error/attack. Once this perimeter is established, undoing the effects of those corrupting update operations can largely be automated.

The goal of this research project is to design and implement a repairable distributed system architecture called *Arachne*, which could serve as the foundation for building intrusion-tolerant network-centric systems. In particular, this project focuses on internet services. Upon detecting an error/attack, an internet service built on *Arachne* could first determine the set of input requests that are directly or indirectly dependent on the error/attack, then undo these requests, re-execute some of them if necessary, and eventually completely eliminate the side effects of the error/attack. The entire repair process is fully automated. Moreover, this fast repairability could be added to the internet services in a way completely transparent to their developers.

2 Technical Approach

Modern web-based internet services typically assume a three-tier architecture, in which a front-end web server, such as Apache or IIS, interacts with the end users, an application server, such as JBoss or Websphere, implements the business application logic specific to an internet service, and a relational DBMS server such as PostgreSQL or Oracle, supports storage and retrieval of persistent data. In some cases, processes running on an application server can also use plain files to store persistent data. In adding fast repairability to internet services, *Arachne* makes the following assumptions about them:

- An internet service's persistent state is stored in either a file system or a DBMS.
- It is acceptable to undo a request whose response has already been returned to the requesting client.
- The response of an internet service to an incoming request depends only on the input request and the service's state.

Because of the first assumption, *Arachne* only needs to ensure updates to file system and DBMS be undoable. The second and third assumptions allow *Arachne* to undo and re-execute innocent requests that are the collateral damage of an error/attack.

The main technical challenge of building repairable systems is how to add fast repairability to existing information systems without requiring any modifications to their internal implementations. More specifically, given an internet service that is based on the three-tier architecture, how to maintain the before image of every persistent state update operation, how to associate the service's persistent state updates with the input requests triggering them, and how to determine whether one request is dependent on another, all without changing the way the service is implemented.

Arachne includes an API-aware compiler that can augment the source code of the set of programs composing an internet service with additional code that maintains sufficient auxiliary information to accurately keep track of inter-request dependencies, and logs all the input requests so that they can be re-executed in the future if necessary. Because communicating components of an internet service could reside in different machines, *Arachne*'s request tracking mechanism also follows messages being exchanged across the network. To do this, the compiler needs to be aware of the APIs used for inter-component communication as well as persistent state modifications. *Arachne* embeds the additional instrumentation code to an internet service transparently and automatically without requiring any involvement from its developers.

To undo file system update operations, the file system image needs to be continuously snapshotted, i.e., every file update operation generates a new version of the updated file block(s). None of the mainstream operating systems, Microsoft Windows OS, Linux, or BSD Unix, support such continuous snapshotting capability. To maximize its portability across multiple platforms, *Arachne* adopts a user-level NFS proxy architecture to implements continuous snapshotting in a way that is independent of the underlying operating system and yet is as efficient as kernel-level implementation. This user-level NFS proxy is able to selectively erase the side effects of those file update operations that should be undone at repair time by issuing corresponding compensating NFS commands.

To undo DBMS update operations, *Arachne* exploits the fact that modern database managers already include a transaction log that records sufficient information for undoing committed transactions. As long as *Arachne* can configure the underlying DBMS to keep its transaction log for as long a period as the fault detection window, no additional database update logging is required. Fortunately, the maximum transaction log size in all commercial DBMSs, including Oracle, Sybase, DB2, and MS SQL server, is indeed configurable. To capture inter-request dependency, *Arachne* adopts a SQL rewriting proxy architecture that intercepts incoming SQL queries/responses, rewrites them to maintain additional bookkeeping information, and extracts these bookkeeping information from SQL query responses to deduce inter-request dependencies. Because this SQL rewriting proxy operates on standardized SQL queries and responses, it is portable across all SQL-based relational DBMS servers on the market. At repair time, *Arachne* undoes each committed transaction that it determines should be aborted through a corresponding compensating transaction. These undo operations are also portable because they too are SQL-based. However, because the format of database transaction log is proprietary, parsing a DBMS's transaction log to construct a compensating transaction for every already committed transaction that needs to be aborted is DBMS-specific.

In summary, given an internet service, *Arachne* automatically instruments its application components to keep track of inter-request dependencies and associations between incoming requests and persistent state update operations. In addition, *Arachne* keeps proper update logs to ensure every file system and DBMS update operation is undoable. At repair time, *Arachne* derives the set of requests that are directly or indirectly

dependent on the error/attack request in question, initiates file system and DBMS undo operations to erase the side effects of these requests, and finally re-executes them according to their timestamps to re-incorporate their effects into the state of the internet service.

Architecturally, *Arachne* consists of three components: (1) an API-aware compiler that can automatically transforms application programs running on the web server and application server into a form that could accurately capture the inter-dependencies among input requests, (2) a portable user-level continuous snapshotting NFS-snooping proxy that could capture dependencies among NFS requests and render every file update operation undoable, and (3) a portable user-level SQL-rewriting proxy that can keep track of inter-transaction dependencies among generate compensating transactions for committed transactions. Combined together, the three components of *Arachne* automate the entire post-intrusion damage repair process, and thus greatly reduce an internet service's mean time to repair (MTTR) after an attack. As a result, *Arachne* is able to provide fast repairability to internet services without requiring any human intervention or software/configuration modifications to the operating system or the DBMS. In other words, *Arachne* adds fast repairability to internet services in a way that is *automatic*, *comprehensive*, and *portable*.

3 Impacts

Although *Arachne* is designed for the three-tier architecture, it is equally applicable to other system architectures such as wide-area-network Web Services and Joint Battlespace Infosphere (JBI), which is built on a publish-subscribe communication mechanism. As for research impacts, the proposed project is expected to make the following contributions to intrusion-tolerant and survivable systems research:

- An automatic message tagging technique that can effectively associate the file I/O operations and database transactions which take place in various components of an internet service with the corresponding input request triggering them,
- A user-level continuous snapshotting file system architecture that can preserve the before image of every file system update operation in a way that is independent of the underlying file system,
- A portable SQL-rewriting proxy architecture that can transparently rewrite SQL queries on the fly to capture inter-transaction dependencies in a way that is both efficient and accurate, i.e., no false negatives and false positives, and
- A reusable implementation framework that could keep track of inter-request dependencies that arise due to shared data that reside in a file system, a DBMS, an address space or communication messages.